Deferred cleanup and error handling in C

Jens Gustedt, Robert C Seacord

Deferred cleanup and error handling in C

Jens Gustedt, Robert C. Seacord

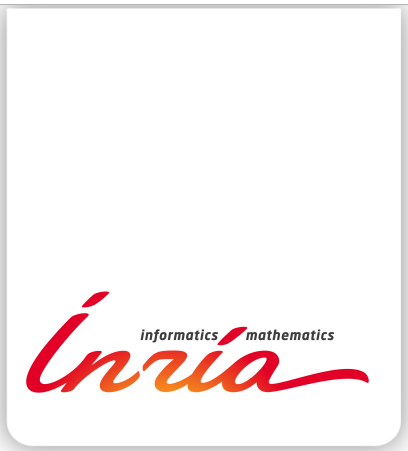
RESEARCH REPORT

N° 9385

Dec 2020

Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--9385--FR+ENG



Deferred cleanup and error handling in C

Jens Gustedt, Robert C. Seacord

Project-Team Camus

Research Report n° 9385 — Dec 2020 — 19 pages

Abstract: This paper introduces the implementation of a C language mechanism for error handling and deferred cleanup adapted from similar features in the Go programming language. This mechanism improves the proximity, visibility, maintainability, robustness, and security of cleanup and error handling over existing language features. This feature is under consideration for inclusion in the C Standard. The library implementation of the features described by this paper is publicly available under an Open Source License at `https://gustedt.gitlabpages.inria.fr/defer/`.

Key-words: C programming language, deferred execution, resource management, error handling, panic-recover

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Nettoyage et traitement d'erreur différés pour le langage C

Résumé : Ce papier introduit l'implémentation d'un mécanisme en langage C pour le traitement d'erreur et le nettoyage différé qui est motivé par une fonctionnalité similaire en langage Go. Il améliore la proximité, la visibilité, l'entretien, la robustesse et la sécurité du nettoyage et du traitement d'erreur comparé à d'autres fonctionnalités du langage. Cette fonctionnalité est en considération pour l'inclusion dans le standard C. L'implémentation des fonctionnalités décrites par ce papier en forme de bibliothèque est accessible publiquement sous une licence libre à https://gustedt.gitlabpages.inria.fr/defer/.

Mots-clés : langage de programmation C, exécution différée, gestion de ressources, traitement d'erreur, panique-recupération

1. Introduction and overview

The defer mechanism can restore a previously known property or invariant that is altered during the processing of a code block. The defer mechanism is useful for paired operations, where one operation is performed at the start of a code block and the paired operation is performed before exiting the block. Because blocks can be exited in multiple locations using a variety of mechanisms, operations are frequently paired incorrectly. The defer mechanism in `C` is intended to improve the proper pairing of these operations. This pattern is common in resource management, synchronization, and outputting balanced strings (e.g., parentheses or HTML tags). Additionally, a panic/recover mechanism allows error handling at a distance.

Most existing high-level languages such as `C++`, `C#`, `Java`, `Go`, `D`, and others include a general mechanism for resource management and error handling. `C` lacks such a mechanism, outside of its use of integer error codes and outmoded mechanisms such as `errno`. Consequently, resource management in `C` programs can be complex and error prone, particularly when a program acquires multiple resources. Each acquisition can fail, and resources must be released to prevent leaking. If the first resource acquisition fails, no cleanup is necessary, because no resources have been allocated. However, if the second resource cannot be acquired, the first resource needs to be released. Similarly, if the third resource cannot be acquired, the second and first resources need to be released, and so forth. This pattern results in duplicate cleanup code and be error-prone because of this duplication and the associated complexity.

`C` programmers need to manage the acquisition and release of resources. Because resources exist in limited quantities, it is always possible that a resource cannot be acquired because the supply of that resource has been exhausted. Examples of `C` standard library functions ISO (2018) that acquire resources include:

- storage: `malloc`, `calloc`, `realloc`, `aligned_alloc`
- strings: `strdup`, `strndup`
- streams: `fopen`, `freopen`
- temporary file: `tmpfile`
- threads: `thrd_create`
- thread specific storage: `tss_create`
- condition variable: `cnd_init`
- condition variable: `cnd_wait`
- mutexes: `mtx_init`, `mtx_lock`, `mtx_timedlock`, `mtx_trylock`

Improper resource management and error handling frequently results in software vulnerabilities. Software security typically assumes an intelligent adversary that is working to compromise the security or possibly the availability of a system. A denial-of-service (DoS) attack occurs when legitimate users are unable to access information systems, devices, or other network resources resulting from the actions of an adversary Mirkovic et al. (2004). DoS attacks attempts frequently take the form of a resource-exhaustion attack that makes a computer resource unavailable or insufficiently available to the application. For example, if an attacker can identify an external action that causes memory to be allocated but not freed, memory can eventually be exhausted. Once memory is exhausted, additional allocations fail, and the application is unable to process valid user requests. Another variant of such erroneous resource management can be exploited in multi-threaded programs that use mutexes. By default, most systems have no provisions to cope with a mutex that is locked by a thread that exits. Other threads that try to access that same mutex will block, eventually causing a deadlock of the entire execution.

Listing 1. An example with three deferred statements

```
guard {
  void * const p = malloc(25);
  if (!p) break;
  defer free(p);
  void * const q = malloc(25);
  if (!q) break;
  defer free(q);
  if (mtx_lock(&mut)==thrd_error) break;
  defer mtx_unlock(&mut);
  // all resources acquired
}
```

Another common error associated with manual memory management is the deallocation of memory more than once, without an intervening allocation. *Double Free* vulnerabilities can be exploited to execute arbitrary code with the permissions of a vulnerable process Seacord (2013). A common source of this error is the deallocation of memory while handling an error condition which is then deallocated again during normal cleanup procedures.

To cope with these shortcomings, the `C` standards committee (ISO/IEC JTC1/SC22/WG14) is investigating the adaptation of `Go`'s mechanism of deferred execution to `C` Ballman et al. (2020). This paper describes the feature including open design choices. The appealing properties of such an integration are

proximity – visibility – maintainability – robustness – security

There is evidence that this mechanism provides improved cleanup and error handling over existing `C` language features (such as **goto** or `longjmp`) and over features that could be imported from other languages such as `C++`, `C#` or `Java` like constructor/destructor pairings, exception handling, or `finally` blocks, or from compiler extensions such as explicit destructor functions (`GCC` compilers).

This paper describes a fully functional implementation of a defer mechanism for `C` that can be used without any compiler specific magic. In particular, it only uses already established features of `C` including nested **for**-loops, `setjmp`/`longjmp`, `volatile` qualifiers, and redefinition of standard features by macros. In addition, this implementation uses some common extensions, to reduce the use of `setjmp`/`longjmp` for unwinding across function boundaries and to avoid the use `volatile` qualifiers. Providing a complete implementation required fixing on a set of design options appropriate for a library only implementation. The final technical specification may eventually make other choices.

The following Section 2 introduces the feature set in more detail and explains these choices. In Section 3, we describe our implementation and its specific properties in more detail. We also describe possible uses of compiler specific properties for improvements and extensions. Section 4 provides a concise overview over the principal features as they are proposed.

2. Description

2.1. **An example for static cleanup handling.** Listing 1 shows a *guarded block* containing three *deferred statements*.

The **defer** keyword indicates that the evaluation of the following statement, such as a call to `free`, is deferred to the end of the guarded block. The deferred statement is evaluated regardless of how the guarded block exits. In Listing 1, the

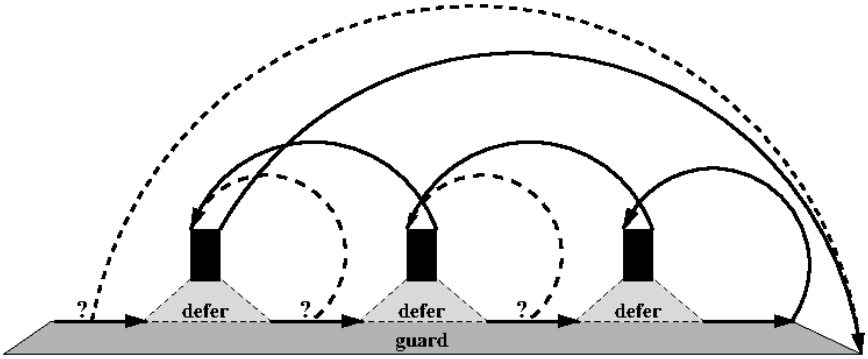


FIGURE 1. Control flow of a guarded block with three defers

block can be exited after the final statement is evaluated, or if a **break** statement is evaluated. Deferred statements are evaluated in the inverse order in which they were encountered. Possible branches for this guarded block are shown in Figure 1. Dashed lines represent conditional error handling paths that are executed when resources are unavailable or evaluation is interrupted by a signal.

This approach has advantages over familiar `C`, `C++`, `C#` or `Java` solutions.

proximity: Cleanup code that releases resources (e.g., `free` or `mtx_unlock`) is collocated with the code that acquires these resources, making it easier for programmers to ensure statements are properly paired.

Other than `C` or `C++` constructs, it also shares another advantage with `C#` or `Java`'s **finally** blocks.

visibility: The cleanup code is clearly visible. This differs from `atexit` handlers or constructor/destructor pairs in `C++`, where cleanup code may be defined in a different translation unit.

For control flow that does not include **return**, `exit`, or other non-returning functions calls an equivalent control flow can be implemented with existing `C` language features. The guarded statement from Listing 1 is equivalent to the code segment in Listing 2. The **if**`(`**false**`)` statement guarantees that the deferred statements are not evaluated when they are first encountered. The labels and **goto** statements implement the backward branches to execute the deferred statements when the guarded block terminates.

A common `C` idiom for cleanup handling is to *linearize* resource management as shown in Listing 3.

This code has the advantage of making the conditional error handling code explicit (with three **goto** statements) but at the cost of proximity as the deallocation code is specified away from the allocations. This linearization requires a naming convention for the labels. For more complicated code the maintenance of these jumps can be error prone. Using **defer** instead eliminates the need for the programmer to define labels.

maintainability: The cleanup code does not depend on arbitrary label names (`C`) or RAII classes (`C++`) and does not change when **defer** or **break** statements are added or removed.

All exits from a guarded block by **break**, **return**, `thrd_exit`, `exit`, or an interruption by a signal must be detected and acted upon. This is difficult to implement in `C` and requires **try/catch** blocks in `C++`. This feature promotes a fourth important property.

LISTING 2. Emulation of **defer** by **goto**

```
{
   void * const p = malloc(25);
   if (!p) goto DEFER0;
   if (false) {
     DEFER1: free(p); goto DEFER0;
   }
   void * const q = malloc(25);
   if (!q) goto DEFER1;
   if (false) {
     DEFER2: free(q); goto DEFER1;
   }
   if (mtx_lock(&mut)==thrd_error) goto DEFER2;
   if (false) {
     DEFER3: mtx_unlock(&mut); goto DEFER2;
   }
   // all resources acquired

   goto DEFER3;
   DEFER0:;
}
```

LISTING 3. A linearization

```
{
   void * const p = malloc(25);
   if (!p) goto DEFER0;
   void * const q = malloc(25);
   if (!q) goto DEFER1;
   if (mtx_lock(&mut)==thrd_error) goto DEFER2;
   // all resources acquired
   mtx_unlock(&mut);
 DEFER2: free(q);
 DEFER1: free(p);
 DEFER0:;
}
```

robustness: Any deferred statement is guaranteed to be executed, provided program execution progresses.

This differs from C++'s handling of destructors, which are only guaranteed to be executed within a **try/catch** block.

For maintainability and robustness, we opted for objects to be accessed by reference within a deferred statement and not by value. Because **defer** is considered and has a similar syntax as other control structures, deferred statements or blocks should not be different from other blocks such as for **if** or **for**. Listing 1 illustrates why an access by reference is key to a successful usage. For example, the value of `p` could be changed by means of `realloc`. Then, if **defer** transfers the captured value to the deferred statement two errors would occur:

- The deferred call to `free` that uses the old (captured) value would be invalid.
- A new deferred statement with the new value of `p` would have to be added.

All of this is simpler if the object is evaluated only when the deferred statement is executed. Because the scope in which the deferred statement is executed is limited, compilers that implement **defer** natively can easily detect and diagnose a usage of a variable that is out of scope. Even for our macro/library based implementation compilers are able to reliably detect the misuse of local variables in deferred statements.

As stated earlier, vulnerabilities can occur if resources are not released, or released more than once. The improved usability of **defer** over existing mechanisms should help eliminated common errors that can result in vulnerabilities and improve security.

security: Resource leaks, deadlocks and double free vulnerabilities are reduced because of increased proximity and visibility of cleanup code to the resource acquisition code.

2.2. **Design choices.** The position paper Ballman et al. (2020) discusses several open design options. One aim of our paper is to evaluate these options in view of a fully functional implementation. This paper here assigns choices to three of of these design options and describes a functioning implementation by means of exiting `C` features.

2.2.1. *Object access.* A deferred statement has to have access to a set of local variables that holds the state in which the cleanup is performed. The main design choices are here to access a variable *by copy* (copied when the **defer** is encountered) or *by reference* (when the deferred statement is evaluated). As previously discussed, we have opted for access by reference, but we also propose an specialized tool **defer_capture** to copy values when the statement is encountered.

2.2.2. *Dynamicity.* In practical examples the need to handle deferred statements dynamically (e.g., in **if** or **for** statements) arises. Unfortunately, dynamic handling of control flow requires that memory be allocated to maintain this state, and these allocations can also fail. We have opted for dynamic handling but we also provide a guarantee that a failure of resource allocation by the **defer** mechanism itself can be handled effectively by applications.

2.2.3. *Scope.* For the Go language, deferred statements are bound to the scope of a function body. For `C`, the need of declaring external functions could severely reduce the expressiveness of the feature, and so a *guarded* block indicated by a **guard** keyword has been introduced.

2.3. **Dynamic vs. static control flow.** Deferred statements are not evaluated when they are encountered, but only after the guarded block terminates. The order and number of times each deferred statement is executed can be made dependent from the context in which the **defer** occurs. For example, if the **defer** statement appears within a loop, a possible design choice is to execute the deferred statement as many times as the loop body is executed, only once, or to forbid the occurrence in such a context completely. Similarly, **defer** that are only executed conditionally because they are within an **if**, **else** or **switch** could be executed in the order they are met, in inverse lexicographic order (which might not be the same because of **goto**), or, again, not allowed in such contexts.

Consequently, the first important design choice is to decide if the evaluation of deferred statements is determined statically at compile time or dynamically at run-time. Making this determination statically at compile time may result in more efficient execution, while making this determination at run-time is potentially less surprising to most programmers.

LISTING 4. Conditional **defer**

```
// create a local buffer as a compound literal
double * p = (len>MLEN) ? 0 : (double[MLEN]){0};

// if len is too big, allocate from the heap
if (p) {
   p = defer_calloc(len, sizeof double);
   defer free(p);
}
// p is a valid pointer until the scope is left
```

Listing 4 shows a code fragment from within a guarded block. The deferred statement will be evaluated at the termination of the guarded block only in cases where the buffer referenced by `p` is dynamically allocated. To support such constructs, our implementation determines the order and number of times each deferred statement is executed at run-time. This approach requires allocating storage at run-time to maintain defer state, in this case at least a flag that holds the information to know if the deferred statement has to be executed at the end of the **guard** or not. If we allow as many executions of the deferred statements as are dynamically encountered, more state must be kept, but generally the state will fit into a small number of bytes, some integers, or pointers. Our implemented defer mechanism provides **fault tolerance** by continuing to operate properly in the event of the failure of a **defer** statement by ensuring the deferred statement is executed to release the resource.

For example, in Listing 4 `defer_calloc` and the **defer** statement may both exhaust memory. If the call succeeds (and a resource is allocated) but the **defer** mechanism fails (because the **defer**-internal state then exhausts memory), the call to `free` is evaluated immediately and the execution of the enclosing guarded block is terminated by a panic with an error code of `DEFER_ENOMEM`.

2.4. **Panic/Recover.** Error handling at distance is supported by the introduction of a panic/recover mechanism, which is similar to `throw`/`catch` in `C++`. Panic/recover depends on the defer mechanism to release resources during stack unwinding. Generally, a *panic* is a condition which requires unwinding one or several levels of deferred statements. Such a panic may occur implicitly if the run-time library detects or signals a fault, or explicitly by the use of a macro dedicated to that purpose.

The `panic` macro is called to indicate an abnormal execution condition. It triggers the execution of all active deferred statements of the current thread in the reverse order they are encountered, until either a deferred call to `recover` is executed or all deferred statements have been executed. If no recover statement is encountered, the function stack unwinds the caller's stack and executes all deferred statements registered in that stack frame. This process continues until a recover expression is encountered or all deferred statements have executed.

Our reference implementation applies the convention that user error codes passed to `panic` are always positive, and system error codes are generally negated `errno` numbers. To deal with such error codes in a portable way, equivalent POSIX error codes are provided with with a `DEFER_` prefix.

System signal numbers cannot be use directly because they may conflict with `errno` numbers. Instead, all POSIX signal numbers have a constant that replaces

the SIG prefix by a DEFER_ prefix. For example, the DEFER_HUP constant represents the signal SIGHUP.

Two convenience macros simplify the use of this feature. The **defer_if** macro is a pseudo selection statement and the recover_signal macro only acts on signals.

Once a deferred statement begins evaluation, the condition that led there can be investigated by means of recover. The recover function returns an integer value indicating why the deferred statement is executing. A return value of 0 indicates that the deferred statement is the result of a **break**, **return**, exit or similar. Any other value indicates that the deferred statement is executing as the result of a panic.

Handling the recovered error is the responsibility of the programmer. The programmer may re-issue the panic from within the deferred statement if it is impossible to recover at a particular level of abstraction.

2.5. **Guarded blocks.** In the proposal to the C standards committee Ballman et al. (2020) each function body implements a guarded block, such that an explicit **guard** statement is not necessary for many common use cases. Generally, this cannot be performed by macros but requires compiler magic. Consequently, eliminating the **guard** statement for the entire function body cannot be provided by a library-only implementation; a library-only implementation will always need a **guard** statement or similar feature that marks the context for a **defer**. Using the implementation specific function __builtin_frame_address our implementation can provide this feature for the GCC family of compilers by keeping track of the stack pointer, and other compilers that provide access to the stack pointer could do similarly.

Beyond the possibility of a library-only implementation, a **guard** keyword allows nesting guarded blocks without having to move inner blocks into their own functions. Go does not require nested constructs for this purpose, because lambdas provide a simple way to define an anonymous functions that can be used for encapsulation. In this paper, we use the phrase *guarded block*, regardless if a **guard** statement is given explicitly or if the function body provides the guarded block.

2.6. **Augmented C library functions.** Existing resource acquisition functions, including those defined by the C Standard library, such as malloc, can be augmented to panic if they encounter an out-of-memory condition. That has several benefits:

- Explicit error handling is not repeated at each call site.
- Programmer-installed deferred cleanup statements will be called, for example, to close files or to free memory allocations.
- User code can recover from failures (panics) using recover; improving system robustness.

Inspection of the generated assembly instructions for these functions shows little overhead for the fast execution path. This technique might also provide an alternative to run-time constraints found in the bounds-checked interfaces in Annex K of the C Standard ISO (2018), in a way that preserves the existing function prototypes.

2.7. **Interoperability with C++.** C++ provides destructors and **catch** clauses for releasing resources. The **defer** statement combines those two features into one.

This implementation provides features to translate between these mechanisms such that stack unwinding can be reliably performed in programs that mix C and C++. In particular, if called from C++ code, the C feature translates panics and requests for exit at the boundary into C++ exceptions. For this to work, C++ applications must establish a mark that indicates the current language is C++. When the C++ code includes <stddefer_codes.h> this is performed for any translation unit that contains the function main.

LISTING 5. `main` as a **try/catch** block

```
int main(int argc, char * argv[]) try {
  // perform operations
} catch (...) {
  throw;
}
```

LISTING 6. A wrapper to translate exceptions to `panic`

```
#include "stddefer_codes.h"
#include "stddefer_cpp.h++"

int testing_CPP(int rec) {
  // boundary must be the first declared variable
  std::defer_boundary boundary;
  try {
    return testing(rec);
  }
  catch (...) {
    boundary.panic();
  }
  return 0;
}
```

Nevertheless, to ensure that resources are released by such a `C++` program it is a good idea to catch all exceptions. This ensures that all destructors down to and including in `main` are called when the stack is unwound. An easy way to achieve this is to establish `main` as a **try-catch** block as shown in Listing 5. This also works for functions that are called with their own threads.

Interoperability for `C++` code that might be **called** from `C` requires that a wrapper `testing_CPP` for a function `testing` is established, as shown in Listing 6.

To properly interoperate when invoked by a `C` program, the `C++` program must:

- declare a local variable of type `std::defer_boundary` for which the constructor records the state prior to the call,
- implement a **try/catch** clause that guarantees that exceptions are caught and that all the destructors are called
- a call to the method `panic` that either propagates any caught exception (if the caller is also `C++`) or that translates an exception to a panic (if the caller is `C`).

Automatic generation of wrappers has not yet been implemented.

Bidirectional translation between `C` **defer** and `C++` exceptions is supported through these mechanisms. To improve the usability of this mechanism, the **defer** error codes are translated into standard `C++` exceptions as shown in Table 1. In contrast to other values, an error code of `0` does not correspond to a panic but to a regular termination as by `thrd_exit` or `exit`.

3. IMPLEMENTATION DETAILS

The important interfaces of this tool are:

- **guard** prefixes a guarded block
- **defer** prefixes a defer clause

TABLE 1. C error codes and C++ exceptions

error code (C or POSIX)	exception (C++)
0	std::defer_exception(0)
-DEFER_EBADF	std::ios_base::failure
-DEFER_EDOM	std::domain_error
-DEFER_EINVAL	std::invalid_argument
-DEFER_ENOMEM	std::bad_alloc
-DEFER_EOVERFLOW	std::overflow_error
-DEFER_ERANGE	std::out_of_range
-DEFER_EUNDERFLOW	std::underflow_error
other	std::defer_exception(other)

- **break** ends a guarded block and executes all its defer clauses
- **return** unwinds all guarded blocks of the current function and returns to the caller
- `thrd_exit` unwinds all active deferred statements of the current thread and exits that thread
- `exit` unwinds all active deferred statements of the current thread and exits the execution[1]
- `panic` starts global unwinding of all guarded blocks
- `recover` stops a panic and provides an error code

The functionality of these interfaces is further detailed in Section 4. The existing features **break**, **return**, `thrd_exit` and `exit` gain new functionality; four principal new features are added to the set, namely **guard**, **defer**, `panic` and `recover`.

The former are overloaded by macro definitions that take effect as soon as the library header is included.[2] Additionally, when linked to legacy object files, C library functions should be used consistently. If the platform supports it, this can be achieved by providing `--wrap=exit` or similar arguments to the linker.

3.1. **Preprocessing language extensions.** The reference implementation is unconventional because it uses preprocessor macros that result in deeply nested **for** loops to implement the principal macros **defer** and **guard**. It does so to ensure a mostly library implementation through macros, that should work with any C compiler. This technique leads to code that is difficult to read and consequently this technique is not recommended for programmer-written code. This technique hides relatively complicated code behind a single keyword (such as **guard** or **defer**) that can also be used as a prefix to another statement. Use of the preprocessor allows for the declaration of local state variables, to perform actions before or after the dependent statement, and to conditionally select an execution or to skip it when necessary.

An intermediate processor is used for the production of the C (and C++) code for the library itself called shnell. The intermediate processor uses **#pragma** annotations for the definition of complicated macros and for *code unrolling*. The distribution also contains the expanded sources and users don't need to perform this processing to use the feature.

[1] For C threads, the behavior when terminating an execution that still has several active threads is undefined.

[2] The changes to the library functions may be suppressed by providing the environment variable `DEFER_NO_WRAP` to the build.

The techniques used in this implementation (but for the omission of the top level **guard** as previously mentioned) show that the semantics that this library-only implementation provides are not extending the usual semantics of `C`, and that their translation and optimization are not more challenging than existing `C` code.

3.2. **Using `longjmp` for control flow.** `C` library features `setjmp`/`longjmp` are used as a principal feature to jump to deferred statements within the same function or to unwind the call stack. We distinguish jumps that are known to target the same function and those that are known to jump to another function on the call stack, named `_Defer_shrtjmp` and `_Defer_longjmp`, respectively. The unwind for a **return** will usually consist entirely of short jumps, whereas unwinding of the call stack by `exit` and other means always initiates a long jump.

The guarantees for the state of local variables that `setjmp`/`longjmp` provides are weaker than those needed in our context. Consequently, atomic variables and fences are used to guarantee up-to-date values. If these are not available, this implementation will work with the precautions described for **defer**.

3.3. **Computed goto.** The implementation can distinguish cases where all jumps are finally implemented as long, or platforms where some shortcut for a short jump can be taken. Currently this is only implemented for GCC and related compilers that implement a *computed* **goto**. A computed **goto** allows a programmer to take the address of a label and then go to that address. Such a specialized implementation performs better during unwinding (primarily comprised of simple jumps), but still must track all the guarded blocks and defer clauses with `setjmp` because these could be jumped to from other functions or from signal handlers.

3.4. **Captured Values.** Following existing `C` semantics for control statements implies that object values are accessed when the deferred statements are evaluated and not when the **defer** itself is encountered. This policy appears to be consistent with the expectation of a majority of programmers that we could reach in a poll where out of 387 developers there was a 2:1 preference for the value being read at the time the deferred statements are evaluated (66.9%) rather than when the **defer** statement is encountered (33.1%).

This policy helps to react to changes to the variable during processing. For example, a pointer value that is to be freed could be rewritten by calls to `realloc` to resize the object. Consequently, our implementation follows this policy. The only restriction is that local variables that are used within a deferred statement must be alive when the deferred statement is executed at the end of the guarded block.

If `C` is extended with lambdas a **defer** statement could be redesigned to accept an equivalent syntax

```
defer [&](void) {
  // deferred statements
}
```

Namely, the **defer** statement could receive a lambda with no parameters and executes it when leaving the **guard**. All objects in this case are captured by reference. Such an extended **defer** statement could also capture local objects by copy. In lambda notation this would appear as follows:

```
defer [&, ptr]() {
  // deferred statements
}
```

In this case, a copy of `ptr` is created when the **defer** statement is encountered, and the value of this copy is used when the deferred statement is evaluated, even if the original value has been modified. In our implementation, copy semantics can be achieved by using the **defer_capture** macro:

```
defer_capture(ptr) {
  // deferred statements
}
```

4. Principal features

4.1. **defer Statement.** The **defer** statement ensures the execution of the deferred statement at the end of the guarded block.

```
defer statement
```

Deferred statements may not themselves contain guarded blocks or other defer clauses, and must not call functions that may result in a termination of the the execution other than `panic`. Additionally, such a call to `panic` must only occur **after** the current panic state had been tested with `recover`.

The deferred statement may use any local object that is visible where the **defer** is placed and that is still alive when the deferred statement is executed, that is at the end of the surrounding **guard** or function body. This property is verifiable at compile time, and a violation usually results in an abortion of the compilation. This implementation here puts everything in place, such that a deferred statement uses the last-written value for all objects, but the success of that depends on the presence of some synchronization features.

If such synchronization features are not available, local objects that may change between the **defer** itself and the execution of the deferred statement and that are used in the deferred statement must be declared `volatile` to ensure the latest written value is read. In general, objects used inside a deferred statement should be **const** qualified to use the original value, and `volatile` qualified to use the last written value.

To keep track of the dynamic state of the deferred statements our implementation uses storage allocation functions `calloc` and `free` to maintain a list of defer clauses. In case that `calloc` fails, the defer clause is executed and the entire execution is unwound by means of `panic` and an error argument of `-DEFER_ENOMEM`. With this strategy, it can be guaranteed that all cleanup code for resources that are allocated before the `calloc` failure is evaluated, and that any call to `recover` in a deferred statement finds the execution in a well-defined state.

4.1.1. **defer_capture** *Macro.* The **defer_capture** macro captures object values and defers execution of the deferred statement.

```
defer_capture([id0 [, id1 [, id2 ...]]]) statement
```

The argument list must be empty, or contain a list of objects. These objects are evaluated and their values stored. When the deferred statement is finally executed, address-less local objects with the same name and type (but const-qualified) are placed before the deferred statement, and are initialized with these frozen values.

4.2. **guard Statement.** The **guard** statement marks a block as guarded by the defer mechanism.

```
guard compound-statement
```

If a guarded block is terminated normally or with a **break** or **continue** statement, all deferred statements that have been registered by a **defer** statement are executed in the reverse order in which they were encountered.

There is also a macro **defer_break** that can be used in contexts where **break** or **continue** statements would refer to an inner loop or **switch** statement. Also, **return**, `exit`, `quick_exit` and `thrd_exit` all trigger the execution of deferred statements, up to their respective levels of nesting of guarded blocks.

Other standard means of non-linear control flow out of or into the block (**goto**, `longjmp`, `_Exit`, `abort`), do not invoke that mechanism and may result in memory leaks or other damage when used within such a guarded block. Our implementation provides replacement for some of these language features such as **defer_goto** and `defer_abort`.

4.3. `panic` **Macros.** The `panic` macros unwind the entire call stack and execute all deferred statements in the thread.

```
noreturn void panic(int C);
noreturn void panic(int C, void F(int));
```

After all deferred statement of the current thread are executed in reverse order in which the **defer** statements have been encountered, in the last stack frame that has a **defer** or **guard** statement, `F(C)` is executed. Here `F` is a `defer_handler` and `C` is an error code.

If provided, `defer_handler` is meant to be a function that terminates the execution, either of the current thread or of the whole program. If omitted, `F` defaults to an implementation specific handler. Valid defer handlers are all functions that have the correct signature and that terminate the current thread or the whole execution. Prominent candidates from the `C` library are `exit` and `thrd_exit`; the first allows to stop the whole execution; the latter, the current thread. Using other handlers such as `quick_exit` or `_Exit`) provides the possibility to manage the set of exit handlers evaluated before the program stops.

This unwind chain can be stopped with a `recover` call within one of the evaluated deferred statements (or if a **defer_if** clause is used).

The error code `C` should be negative if it is supposed to map to a system defined error condition (as if for `errno` or a caught signal). Otherwise, all user supplied error numbers should be positive. When unwinding an error condition triggered by the system, the error code will always be negative.

4.3.1. *`defer_assert` Macro.* The `defer_assert` macro asserts a run-time condition or panics.

```
void defer_assert(condition, code, string);
```

The macro `defer_assert` is similar to the `assert` macro in that it asserts that a specific condition holds. It cannot be switched off at compile time but the panic that is triggered can be recovered. In particular, the compiler can detect that the following code can only be reached if the condition holds, and optimize the code accordingly. For example, the following asserts that pointer `p` is non-null, and otherwise triggers a `ENOMEM` panic.

```
defer_assert(p, -DEFER_ENOMEM, "allocation␣failed");
// p is now assumed to be non-null
```

4.4. **Regular termination.** There are three severity levels for termination of a guarded block, **break** or similar just terminates the guarded block, **return** terminates all guarded blocks of the same function and `exit` or `panic` of the same thread across all function calls.

The implementation augments **break** (within a **guard**) and **return** by this functionality. This is achieved by overloading theses features by macros, a technique that that the `C` standard explicitly allows.

For the `C` library functions that terminate a thread or the whole execution, the functionality of the following functions is the same as their unprefixed `C` library counterparts.

```
noreturn void defer_exit(int);
noreturn void defer_thrd_exit(int);
noreturn void defer_quick_exit(int);
noreturn void defer__Exit(int);
```

Instead of terminating the thread or the execution immediately they will unwind the stack of collected deferred statements.

Code that is compiled with the `<stddefer.h>` header will also just replace calls to the `C` library functions

```
noreturn void exit(int);
noreturn void thrd_exit(int);
noreturn void quick_exit(int);
```

by calls to their corresponding wrappers.[3]

Other code that is not compiled with it, may still call the `C` library functions directly without unwinding. Mixed units of different origin should use linker tricks such as `-wrap-function=exit` to replace these calls by the wrapped ones.

4.5. `recover` **Function.** The `recover` function stops a panic and returns an error code.

```
int recover(void);
```

A call to this function must reside within a deferred statement.

This will only stop unwinding of the panic if the error code had not been `0`. Normal **break**, **return**, `thrd_exit`, `exit` etc will not be be stopped and the guarded block, function, thread or program will finish when all the associated deferred statements have been executed.

If the error code is non-zero, execution of the defer clause then continues as if the nearest guarded block had been broken by **break**. That is, the execution of the defer clauses of that particular guarded block are continued and then execution resumes at the end of that guarded block.

In general, error codes provided by the system will be negative and user provided error codes shall be positive. To determine if a system error occurred you may, for example, compare the value to `-ERANGE`, or, if the error originated from a caught signal to `DEFER_SIGNO(SIGTERM)`.

4.5.1. **defer_if** *Macro.* The following **defer_if** macro invocation:

```
defer_if(err) statement-1
else          statement-2
```

[3] A compilation that set the macro `DEFER_NO_WRAP`, will not replace the usage of the `C` library termination functions by the wrapped versions as presented above. The use of this macro is not recommended.

is equivalent to the following defer clause:

```
defer {
    register int const err = recover();
    if (err)
        statement-1
    else
        statement-2
}
```

This stops an unwind originating from a `panic` call or from a signal with non-zero error code (or similar), returns the error code in a local `int` variable named `err`, and executes the depending statement if the value is non-zero, or an alternative **else** clause, if any, if the value is zero. The **else** clause is optional.

The variable `err` can then be used inside the **if** or **else** clauses, but it is not mutable and its address cannot be taken. And it is a bit useless within the **else** clause, because we know that it is zero, there.

4.5.2. *`defer_show` Macro.* The `defer_show` macro shows information about the latest `panic` on `stderr`.

```
void defer_show(int a);
```

This can be used as diagnosis tool after a `recover` to provide feedback to the user what went wrong with the execution.

4.6. **Signals.** The execution of a deferred statement can be triggered in two ways. First, it can be *explicit* by leaving a guarded block when coming to the end of it or by using **break**, `exit` ... from within as discussed above.

Then it can be triggered implicitly from a signal handler. Two signal handlers are provided `defer_sig_flag` and `defer_sig_jump`, and a function `recover_signal` may help to process signals. The provided signal handlers use extensions that might not be available everywhere. In particular, they use the fact that thread-local variables can be accessed from a signal handler. The implementation does not install any of these handlers by default.

```
int recover_signal(void);
```

The function differs from `recover` because it only reacts on signals that are caught by the provided signal handlers (`defer_sig_flag` and `defer_sig_jump`), not on other **break** or `panic` events. Also, the function returns the number of the signal that has been caught, not the raw error code as it is maintained by the library.[4]

It can be used in any context and does not need a surrounding guarded block or **defer** clause. Its cost is the lookup of a thread local variable and a memory synchronization with that variable. So it should probably not be placed inside performance critical loops.

When a signal occurs within a guarded block or defer clause and **is not** recovered by means of `recover` or `recover_signal`, the execution is considered to be compromised, a panic results, and the deferred statements of the thread are unwound.

A typical use of this in conjunction with `defer_sig_flag` would be an active loop, see Listing 7. Here, an `INT` event can occur at any point of the computation before or inside the **while** loop. At the first detection of a signal, the **while** loop

[4]An auxiliary macro `DEFER_SIGNO` can be used to translate between signal numbers and raw error codes.

LISTING 7. A loop that is interruptible by a signal

```
int main(void) {
  // Establish the handler for user interrupts.
  signal(SIGINT, defer_sig_flag);

  while (active) {
      if (recover_signal()) {
          puts("caught signal, stopping loop");
          break;
      }
      ... do whatever you have to do ...
  }
}
```

then is broken, and execution continues after it. This guarantees that the whole loop body is always executed until its end.

The function

```
void defer_sig_flag(int sig);
```

provides a signal handler that flags the execution of a thread and is not too intrusive. It immediately hands control back to the user code and the effective handling of the event is delayed until the application code reaches the end of a guarded block and/or actively issues a `recover_signal` operation to receive the code.

Generally this might be used to handle signals triggered by the user via the terminal. In contrast to that, this function is not suited for signals that jump back to exactly the same code location where some repair work has to be performed to continue, such as invalid arithmetic or segmentation faults. In such situations we need

```
void defer_sig_jump(int sig);
```

a signal handler that flags the execution of a thread and jumps to the first deferred statement, an action that is much more intrusive and that assumes that the signal is caught by the same thread that is to perform that defer clause. Not all platforms may guarantee this.

The `C` standard function `signal` itself is not well suited for this kind of handler, because signals should be switched off during the handling of such interrupts. A given platform (such as POSIX) might propose tools are more appropriate than `signal` to handle such faults.

4.7. **Storage management functions.**

```
void * defer_malloc(size_t size);
void * defer_calloc(size_t nmemb, size_t size);
void * defer_realloc(void * ptr, size_t size);
void * defer_aligned_alloc(size_t alignment, size_t size);
```

These functions behave like their unprefixed `C` library counterparts, only that instead of returning a null pointer they panic on failure. For example, `defer_malloc`, panics with `-DEFER_EINVAL` when called with a `size` of `0` and with `-DEFER_ENOMEM` when the allocation fails.

This approach eliminates the user requirement to check of the result of such a function and subsequent code is guaranteed that the return value is a non-null

Listing 8. An example for a simple buffer reallocation

```
void * p = malloc(small);
defer free(p);
// use p
defer {
    // save some precious values from p
}
p = defer_realloc(p, huge);
```

pointer. For example other than for `realloc`, Listing 8 shows a valid idiom. The assignment to `p` is only executed if the call to `defer_realloc` is successful, and the call to `free` in the deferred statement will always use the good (= last allocated) value for `p`. Also, if it fails execution is immediately transferred to the currently deferred statements, and the necessary terminal actions are taken as indicated, first the second deferred statement may save values from `p` and then `p` is freed.

A compilation that sets the `DEFER_MALLOC_WRAP` macro replaces all usage of the `C` library storage allocation functions by the wrapped versions. By using this feature whole projects can switch to such an error checking policy for storage allocation at once.

5. Conclusion

An effort is underway to add a mechanism for error handling and deferred cleanup to the `C` standard by providing a new library clause for a new header with the proposed name `<stddefer.h>` Ballman et al. (2020). In this paper we have presented a fully functional library implementation of these features, that in its most restricted form only uses established `C` constructs. We have shown that the **defer** concept itself does not extend the semantics of `C`, but provides language constructs to allow programmers to develop maintainable, robust, and secure code.

The library described by this paper is publicly available under a BSD 3-Clause Open Source License. The project page is located at

`https://gustedt.gitlabpages.inria.fr/defer/`.

Acknowledgments. Thanks to Aaron Ballman, Alex Gilding, JeanHeyd Meneide, Miguel Ojeda, Tom Scogland, Martin Uecker, Freek Wiedijk, and Simon Harraghy for their contributions to this work.

References

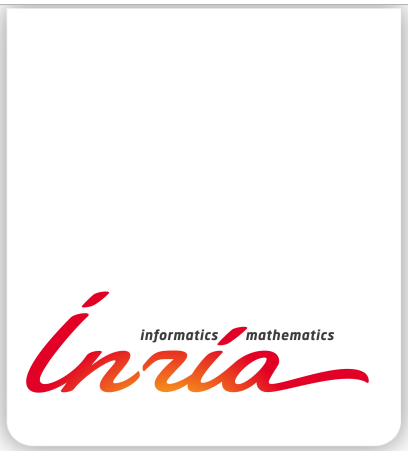
Aaron Ballman, Alex Gilding, Jens Gustedt, Tom Scogland, Robert C. Seacord, Martin Uecker, and Freek Wiedijk. 2020. Defer Mechanism for C. ISO SC22 WG14 N2542, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2542.pdf`.

Jens Gustedt and Robert C. Seacord. 2021. C language mechanism for error handling and deferred cleanup. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC'21)*. ACM, virtual meeting, South Korea, 3 pages. `https://hal.inria.fr/hal-03059076` to appear.

ISO. 2018. *ISO/IEC 9899:2018: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland.

Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. 2004. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall, Upper Saddle River, NJ, USA. 372 pages. Radia Perlman series in computer networking and security.

Robert C. Seacord. 2013. *Secure Coding in C and C++* (2nd ed.). Addison-Wesley Professional.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399